International Journal of Research in Engineering, Science and Management
Volume-2, Issue-4, April-2019
www.ijresm.com | ISSN (Online): 2581-5792

Desktop Data Search for Big Data using Classifier and Indexer Technique

Harshita Shukla[1], Vinod Todwal[2]
[1]*Student, Department of information Technology, Rajasthan College of Engineering for Women, Jaipur, India*
[2]*Assistant Professor, Dept. of information Technology, Rajasthan College of Engg. for Women, Jaipur, India*

***Abstract*: Desktop data search is an important feature for any desktop and is essentially required too. For searching any big data the systems are required to have the search algorithms in order to access the hard disks and the connected storage spaces, sometimes it may get difficult to find the data manually by simply scanning the visible strings, that where the search algorithms comes handy. Search algorithms can be inspired from many technologies but they are needed to be very specific about the task and precise too.**

***Keywords*: Tree Algorithm, Indexer, Searcher, Folder Accession.**

1. Introduction

Desktop search algorithm is an important tool for the current scenario based system, the large data are occupying spaces and are required to get access continuously for these type of function the system requires Desktop Data Search Algorithm.

Many tasks require a programmer to organize data in collections and perform different operations on these collections. Moreover, the collections and the operations must often be designed in a way that guarantees certain parameters of program execution, for example speed and memory consumption. Because working with data collections or sets is so frequently encountered exercise, a number of attempts has been made to standardize these exercises and, thus, reduce the time and effort of their implementation. For this reasons many standard data structures and algorithms appeared. Using these well-defined data structures and algorithms programmers can quickly and efficiently solve various tasks.

One of the standard data structures that has been widely used in programming is the tree data structure. Tree structure means a "branching" relationship between nodes (Knuth, 1973) and imposes a hierarchical structure on the collection of items. There are many types of trees: binary trees, balanced trees, 2-3trees, B-trees, red-black trees, Fibonacci trees, AVL trees to name just a few. Each type of the tree data structure has been designed to support a specific set of properties essential in a given situation.

Tree structures are the most important nonlinear structures that arise in computer algorithms (Knuth, 1973). Trees have numerous applications. They are used to analyze electrical circuits, to represent the structure of mathematical formulas, to organize information in database systems, to present the syntactic structure of source programs in compilers and many others (Aho et al., 1983). Data that is stored in the memory needs to be retrieved. Now, various techniques exist, which can be used to search and retrieve a required element from the data set. The most widely known algorithms used for searching for an element in a given array are Linear Search and Binary Search.

- *Linear Search:* The simplest method of finding out an element from an array would be to visit each element in the dataset sequentially, compare the element with the key element required, and then return the result as found/ not found along with the position if found. This method describes the Linear Search algorithm. Linear Search is also called as Sequential Search.
- *Binary Search:* For an array to be sorted by Binary Search, first and fore mostly it is necessary that the array to be searched is sorted in the ascending order. Once this constraint is satisfied, in order to search for the key value, the algorithm makes use of 3 variables l, r and m which stand for left, right and middle and represent the position of the elements in consideration, and then the key value is compared with the element in the middle of the array. If both the values do not match, then the array is divided into 2 parts, in both of which the middle value is compared to the key value, and if found, the index is returned. If not, this process continues till the required key value is obtained in the array. If the key is not present in the array, a message saying 'element' not present is displayed to the user. Each method has its own problems. Linear search requires a large amount of time for searching, especially if the element is towards the end or in middle of a large data set. Binary search requires data to be stored which takes large time.

2. Literature review

In this section, we will introduce in detail the existing indexing and nearest neighbor (NN) searching algorithm in the literature. As it is a widely studied area and many algorithms have been proposed, we only concentrate on the algorithms suitable to large-scale high-dimensional database. For more information on NN searching area, the survey paper will be recommended [1].

Generally, the existing NN search algorithms in large-scale high-dimensional database can be classified into three categories based on the ways to index data: algorithms based on hierarchical partitioning trees, algorithms based on clustering methods and algorithms based on hash methods [2].

The most representative technique in hierarchical partitioning tree is perhaps the k-d tree. However, as mentioned above, the performance of k-d tree will rapidly decrease with the increase of the dimensions. To alleviate the decrease performance, researchers propose approximate nearest neighbor (ANN) search, and perform limited backtracking steps ordering by the distances between the query point and backtracking nodes. The backtracking process will be stopped if it satisfies a “error-bound” condition or a “time bound” condition [4].

To improve the performance of above k-d structure, Silpa-Anan and Hartley [6] present multiple randomized k-d trees. These randomized k-d trees are built in parallel by randomly selecting some top dimensions with highest variances as split hyperplanes. Compared to the traditional “error-bound” or” time bound” k-d tree, multiple randomized k-d trees will probably obtain higher performance in accuracy and efficiency. Muja and Lowe [2] examine the randomized k-d trees data structure and evaluate it with more exhaustive tests. Their implementation has been incorporated into OpenCV and is now considered as one of the state-of-art nearest neighbor matching algorithms.

Some other algorithms focus on how to find a more optimal split hyperplane or hypersphere rather than randomly selecting the split hyperplane, such as the principal component trees (PCA-trees) [10], the random projection trees (RP-trees) [11] and the improved PCA-trees [12]. They report that their optimal splitting methods will improve the overall performance compared to the classic k-d trees.

Actually, the more adaptive and optimal splitting way is the clustering on the data, which can further reduce the partitioning error. Thus, there are also many NN searching algorithms based on various clustering methods, such as Kmeans trees [13], vantage point trees (VP-trees) [14], cover trees [15], agglomerative clustering trees [6], etc.

Fukunaga and Narendra [16] present the hierarchical K-means tree that clusters data points into K disjoint groups by K-means algorithm, and then recursively performs the same operation on each group until the size of all the groups lower than a given threshold. In query process, a large number of neighbor clusters must be retrieved to maintain high performance as the tree only be traversed once without any backtracking. To alleviate it, Muja and Lowe [2] present to explore the k-means tree by a best-bin-first strategy, which is proved to be a more effective in improving the overall performance.

Performing NN search by product quantization (PQ) approach is another recent research focus. Jegou et al. propose constructing a quantizer of the high-dimensional space as a Cartesian product of lower dimensionality quantizers, called product quantization [17]. Points are represented by the short code composed of the quantization indices of its subparts. Owing to this representation, the query process can be efficiently performed by a look-up table technique. The PQ variants and its improvements include optimized product quantization [18], additive quantization [19], stacked quantization [20] and so on. However, these methods are always implemented in a single machine, which is not suitable to large-scale database.

To perform NN search in high-dimensional data, especially when the dimension is relatively high, a well-known choice is utilizing hash methods which can alleviate the “curse of dimensionality” problem. Among these hash methods, local sensitive hash (LSH) perhaps is the most frequently used [21]. LSH uses a large number of hash functions, which hash nearby points in the original metric space into same buckets with high probability. The query point will take the points located in buckets as NN neighbor candidates.

It is obvious that the LSH index is a kind of flat index. So, LSH must maintain a large number of hash functions to improve the matching performance including both the accuracy and recall rate. The multi-probe LSH approach [7] is proposed to reduce the LSH storage cost by querying the adjacent hash bucket. The amount of the hash functions is reduced by an order of magnitude. To further increase the performance of LSH, Bawa et al. proposed LSH forests which is better adaptive to data and has been successfully applied in text retrieval area [8].

In the case of distributed LSH, Panigrahy proposes Entropy LSH method [22] to significantly reduces the number of required hash tables. To maintain the overall performance, a large number of query offsets must be generated and hashed in the buckets to find new candidates. It will obviously increase the network cost and limit the application of Entropy LSH in the distributed environment. To overcome it, Bahmani et al. proposed a scalable layered LSH [23] that distributed the adjacent hash bucket to the same computing node. They prove that the layered LSH exponentially decreases the network cost, while maintaining a good load balance between different computing nodes. They also give an implementation of the layered LSH on Hadoop parallel framework.

3. Proposed work

Define Database function calls a directory provided by the operating system and registers down the content of it. Content may contain files, folders and sometimes system files as well. The hierarchy can be complicated and may require multiple recursions for registering the content. By stating the keyword hierarchy, it is meant to be understood that a root folder will contain subdirectories and those subdirectories will definitely have separate content of files and folders.

In this algorithm the defined database function registers down every file and folder sequentially with their original locations and keeps updating the newly added files and folders. This maintains a record of the stored files and folders.

MATLAB uses “dir” function to update the list or the database generated by the defined database function. “dir” functions provides various information about the content of the current directory such name, folder name, date, size, directory status and the serial date number for the mathematical calculations. Validation function checks the user’s integrity and only allows the users those are registered or who have the login credentials.

Search algorithm: The algorithm works over various connected files or scripts, and is basically of type Indexer algorithm where the data gets stored and then it is to fetched for searching. Benefits of defining database is it takes less time to search the data as access folder by folder and the sub folders and then looking up for the files will take extra amount of time and will return the data one by one which is a longer process, however the process followed by the proposed algorithm is entirely different from the other search algorithms implemented by the operating systems. Proposed algorithm works over the root to tip Tree algorithm where the searcher or the indexer moves from branch to branch to register down the content and then form the database in the same tree hierarchy. Tree algorithm save a lot of simulation time when implemented through the data register process as searching out the data out of the string is always going to be a swift operation in comparison to the accessing the folder.

4. Results

The proposed algorithm has been tested over thousands of files and has produced results on time with respect to string length.

Table 1
Timing and String Length Comparison

File Type	Number of Files/Folders	String Length (Average)	Time
Excel Files	450	12	0.0261
Folders	406	6.88	0.0234
Bmp Files	357	10.94	0.0234
JPG Files	191	10.87	0.0221
Png Files	499	10.90	0.0269
Word docx Files	176	11.64	0.0254

The above presented table represents the timing and the string length comparison for different type of file search and folder search.

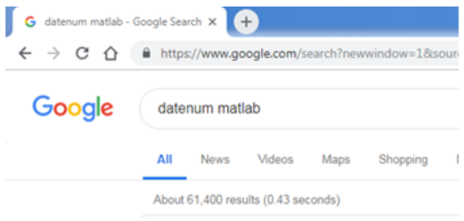
datenum matlab - Google Search
https://www.google.com/search?newwindow=1&sour
Google
datenum matlab
All
News
Videos
Maps
Shopping
About 61,400 results (0.43 seconds)

Fig. 1. Google search result time display

5. Conclusion

The above presented work shows that the data searching in the desktop is tricky and complicated too, but with the help of database functionality and the use of tree search algorithm in the same can help in reducing the timing for the searching of data. The average timing results for searching data shown by the google and the yahoo search engines is around 0.43 seconds. Which can get reduced if the database intellectuality is introduced into the searching algorithms.

References

[1] J. Wang, W. Liu, S. Kumar, S. F. Chang, learning to hash for indexing big data-a survey, Proceedings of the IEEE 104 (1) (2016) 34–57.
[2] M. Muja, D. G. Lowe, Scalable nearest neighbor algorithms for high dimensional data, IEEE Transactions on Pattern Analysis and Machine Intelligence 36 (11) (2014) 2227–2240.
[3] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, A. Y. Wu, An optimal algorithm for approximate nearest neighbor searching fixed dimensions, J. ACM 45 (6) (1998) 891–923.
[4] C. Silpa-Anan, R. Hartley, Optimised kd-trees for fast image descriptor matching, in: 2008 IEEE Conference on Computer Vision and Pattern Recognition, 2008, pp. 1–8.
[5] R. F. Sproull, Refinements to nearest-neighbor searching ink-dimensional trees, Algorithmica 6 (1) (1991) 579–589.
[6] S. Dasgupta, Y. Freund, Random projection trees and low dimensional manifolds, in: Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing, STOC ’08, ACM, New York, NY, USA, 2008, pp. 537–546.
[7] Y. Jia, J. Wang, G. Zeng, H. Zha, X. S. Hua, Optimizing kd-trees for scalable visual descriptor indexing, in: 2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2010, pp. 3392–3399.
[8] D. Nister, H. Stewenius, Scalable recognition with a vocabulary tree, in: 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’06), Vol. 2, 2006, pp. 2161–2168.
[9] P. N. Yianilos, Data structures and algorithms for nearest neighbor search in general metric spaces, in: Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA ’93, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1993, pp. 311–321.
[10] A. Beygelzimer, S. Kakade, J. Langford, Cover trees for nearest neighbor, in: Proceedings of the 23rd International Conference on Machine Learning, ICML ’06, ACM, New York, NY, USA, 2006, pp. 97–104.
[11] K. Fukunaga, P. M. Narendra, A branch and bound algorithm for computing k-nearest neighbors, IEEE Transactions on Computers C24 (7) (1975) 750–753.
[12] H. Jegou, M. Douze, C. Schmid, Product quantization for nearest neighbor search, IEEE Transactions on Pattern Analysis and Machine Intelligence 33 (1) (2011) 117–128.
[13] T. Ge, K. He, Q. Ke, J. Sun, Optimized product quantization for approximate nearest neighbor search, in: 2013 IEEE Conference on Computer Vision and Pattern Recognition, 2013, pp. 2946–2953.
[14] A. Babenko, V. Lempitsky, Additive quantization for extreme vector compression, in: 2014 IEEE Conference on Computer Vision and Pattern Recognition, 2014, pp. 931–938.
[15] J. Martinez, H. H. Hoos, J. J. Little, Stacked quantizers for compositional vector compression, CoRR.
[16] A. Andoni, P. Indyk, Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions, Commun. ACM 51 (1) (2008) 117–122.
[17] Q. Lv, W. Josephson, Z. Wang, M. Charikar, K. Li, Multi-probe lsh: Efficient indexing for high-dimensional similarity search, in: Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB ’07, VLDB Endowment, 2007, pp. 950–961.
[18] M. Bawa, T. Condie, P. Ganesan, Lsh forest: Self-tuning indexes for similarity search, in: Proceedings of the 14th International Conference on World Wide Web, WWW ’05, ACM, New York, NY, USA, 2005, pp. 651–660.
[19] R. Panigrahy, Entropy based nearest neighbor search in high dimensions, in: Proceedings of the Seventeenth Annual ACM-SIAM Symposium on

Discrete Algorithm, SODA ’06, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006, pp. 1186–1195.

[20] B. Bahmani, A. Goel, R. Shinde, Efficient distributed locality sensitive hashing, in: Proceedings of the 21st ACM International Conference on Information and Knowledge Management, CIKM ’12, ACM, New York, NY, USA, 2012, pp. 2174–2178.